A Blockchain Protocol for Human-in-the-Loop AI

Nassim Dehouche*
Mahidol University International College
Mahidol University
Salaya, Thailand 73170
nassim.deh@mahidol.edu

Richard Blythman
Algovera
Dublin
Ireland
richard@algovera.ai

Abstract

Intelligent human inputs are required both in the training and operation of AI systems, and within the governance of blockchain systems and decentralized autonomous organizations (DAOs). This paper presents a formal definition of Human Intelligence Primitives (HIPs), and describes the design and implementation of an Ethereum protocol for their on-chain collection, modeling, and integration in machine learning workflows.

1 Introduction and Related Work

Modern Artificial Intelligence tends to focus on centralization, autonomy and competition with humans (1). However, the idea of augmenting human intelligence (2) and "man-computer symbiosis" (3) was prevalent in the early days of AI and cybernetics.

Human-in-the-loop (HITL) machine learning (4) is a promising development in this regard. Intelligent human inputs are often included in the machine learning workflow before training, in the form of data annotation. The HITL approach extends the scope of this integration to include human-machine interactions during training, e.g. through expert supervision (5), and post-training, e.g. in safety audits and fine-tuning models (6).

Software applications for crowdsourcing human intelligence tasks face challenges pertaining to the unfair compensation of labor (7), fraud (8), censorship (9), and the difficulty of vetting credentials (10). The latter is typified by protocols geared towards centralized crowd-labor platforms, such as Turkit (11). For example, human input is taken from an indistinct mass of crowd workers on Amazon's Mechanical Turk platform, without the ability to require a certain level of expertise or credentials from respondents. Turkit (11) introduced the useful concept of scripting human intelligence tasks within traditional web applications, and was designed with issues related to high-cost and high-latency steps involving humans in mind. This required engineering a *crash-rerun* approach to avoid re-executing expensive steps.

Decentralized software deployed on public, permissionless blockchains offers natural opportunities to tackle the aforementioned challenges. Any write instruction in a smart contract is an atomic transaction that is immutably stored on the blockchain, and transparently accessible to any client application. Moreover, in addition to trustless, uncensorable payment processing, blockchain software can offer participants ownership in the system they partake in. Lastly, the emergence of standards for identity management, such as the non-fungible token standard, allow for sophisticated access control and have propelled the emergence of domain-expert decentralized autonomous organizations (DAOs).

DAOs are sometimes imagined as being governed by autonomous algorithms, with humans at the margins. However, there is an increasing push towards a future of collective intelligence that promotes harmony between humans and algorithms by optimizing for the autonomy of individuals (12). We

*https://www.ndehouche.github.io

2022 Trustworthy and Socially Responsible Machine Learning (TSRML 2022) co-located with NeurIPS 2022.

believe that protocols that facilitate crowdsourcing of human intelligence and preferences are a key component of this. This has applications in collection and annotation of training data, and AI safety.

In the following, we describe a protocol for Ethereum Virtual Machine (EVM)-compatible blockchains that allow for the on-chain modeling of human intelligence tasks and their integration in machine learning workflows.

2 Human Intelligence Primitives

A Human Intelligence Primitive (HIP) is a procedure for the collection and representation of preferences structures on a finite set of n potential alternatives (i.e. comparable objects or actions) $A = \{a_0, \dots, a_{n-1}\}$. Preferences can be of one of the four following types, based on (13):

- A choice (P_1), that is a subset $A' \subseteq A$ of potential alternatives, typically a singleton set, containing the preferred alternative(s).
- A ranking (P_2), that is a total preorder on A, ordering alternatives by decreasing preference, with possible *ex-aequo*. A particular case of ranking is preferential voting (18), in which this preference structure is a total order on A.
- A sorting (P_3), that is the assignment of each alternative in A into pre-defined classes $C = \{c_0, \dots, c_{k-1}\}$, ordered by decreasing preference. A particular case of sorting is score voting (19) on a discrete scale.
- A classification (P_4), that is the assignment of each alternative in A into pre-defined, unordered classes $C = \{c_0, \dots, c_{k-1}\}$.

A HIP can thus be abstractly characterized by a triplet (t, n, k), where $t \in \{P_1, P_2, P_3, P_4\}$ is the type of preferences sought, $n \in \{2, 3, \dots, +\infty\}$ the number of alternatives considered, and $m \in \{1, 2, \dots, +\infty\}$ the number of classes (equal to 1 for a choice or ranking primitive).

3 Smart Contract Architecture

We propose a smart contract implementing HIPs to incentivize and coordinate collective intelligence by humans within DAOs. HIPs can be initiated by Externally-Owned Account (EOA) on Ethereum, or contracts, through a CALL or DELEGATECALL operation. Conversely, the smart contract can communicate synchronous events to off-chain clients (e.g. the submission of a response to a HIP), and its output can be read asynchronously by these programs.

We consider two categories of users of the smart contract; *proposers* and *respondents*. A HIP is recorded in a HIP object, the creation of which is initiated by a *proposer* address, through a function `submitHIP()`. In this function, a proposer submits a triplet (t, n, k), and pays a fee that depends on the type of primitive t. The type t is encoded by an enumerable `types {CHOICE, RANKING, SORTING, CLASSIFICATION}`, while the number of alternatives n and the number of classes k are stored in unsigned integers.

Additionally, the proposer specifies a duration, encoded as an unsigned integer number of seconds, for taking responses. This duration is relative to the creation date of the HIP, recorded as the timestamp of the valid block enacting its creation on the blockchain. Lastly, the HIP object records the number of responses (individual preference structures compatible with the HIP type) recorded so far, in an unsigned integer variable `numResponses`.

An individual response to a HIP is submitted by a *respondent* address, through a function `submitResponse()`, specifying the address of the proposer, and the index of the HIP being responded to, among their proposed HIPs. Respondents' access is gated by a non-fungible token (NFT), vetting their credentials, and giving them *read* access to the corresponding off-chain semantic data, and *write* access to record a response to a HIP in the contract. Moreover, before recording a response, we verify that the respondent has not already voted, and that the submitted response is compatible with the HIP type t. An individual response that passes these checks is recorded in a `Response` structure, containing the address of the respondent and an array of unsigned integers, representing the content of the response.

Given the strengths of the blockchain (trustless access control and payment processing), and its weaknesses (inability to store secrets and high cost of computation), we have made the following key architecture choices:

- Since data are transparently stored on the blockchain, HIP objects are recorded in the abstract form of a triplet (t, n, k), and linked with semantic data (i.e. descriptions for alternatives and classes) that are stored off-chain.
- In order to incentivize responses, and discourage their concentration in a few HIPs, the reward of each respondent is the fee paid by the proposer divided by the number of responses, by the end of a HIP's duration.
- Once recorded in the contract, responses can be eventually accessed by off-chain clients for computationally complex processing and aggregation.

The proposed architecture is summarized in the process diagram in Figure 1.

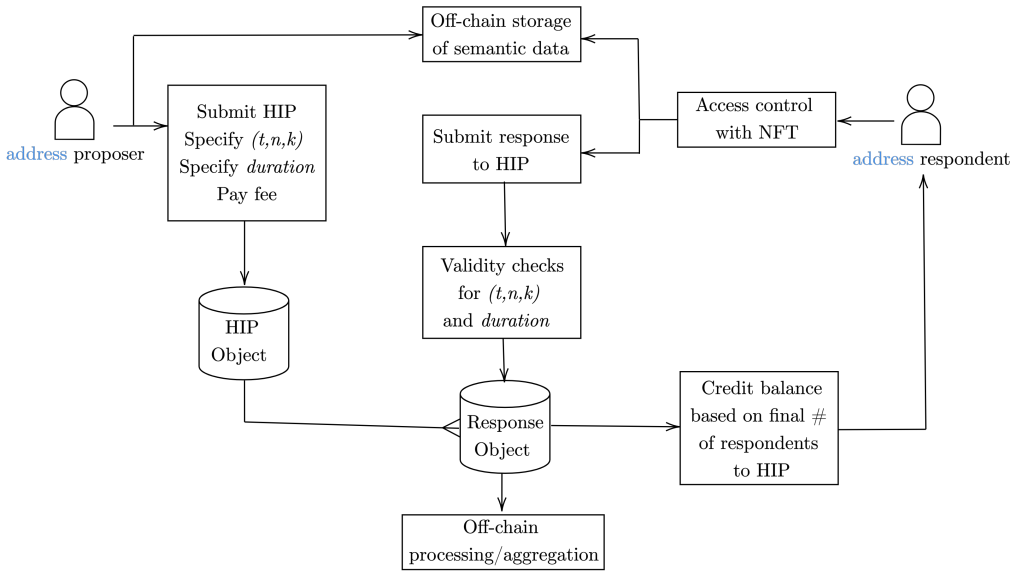

Figure 1: Architecture of the proposed protocol

4 Main Data Structures

The implementation of the protocol is subject to four indexing requirements:

- Reading/Writing HIPs requires mapping proposers with the HIPs they have created. This is implemented as a `mapping(address => HIP[])`, named `HIPs`, whose key is a proposer address, and value is an array of HIPs submitted by this address.
- Reading/Writing Responses requires mapping HIPs with the responses they have received. This is implemented as a double mapping, `mapping(address => mapping(uint => Response[]))`, named `responses`, indexed by a proposer address and an integer index for a HIP, and whose value is an array of responses submitted for it.
- Ensuring single responses requires mapping respondents and HIPs, with a boolean indicating whether the former has submitted a response to the latter. This is implemented as a triple mapping, *mapping(address => mapping(address => mapping (uint =>bool)))*, named `responded`, indexed by a respondent address, a proposer address and an integer index for a HIP, and whose value is a boolean indicating the existence of a response.

- Payment processing requires mapping respondents with the proposers and indices of the HIPs they have responded to. This is implemented as a mapping, `mapping(address => ResponseRef[])`, named `responseRefs`, indexed by a respondent address, and whose value is an array of objects of type `responseRef`, containing the address of a proposer and the index of a HIP.

These four mappings are illustrated in Figure 2.

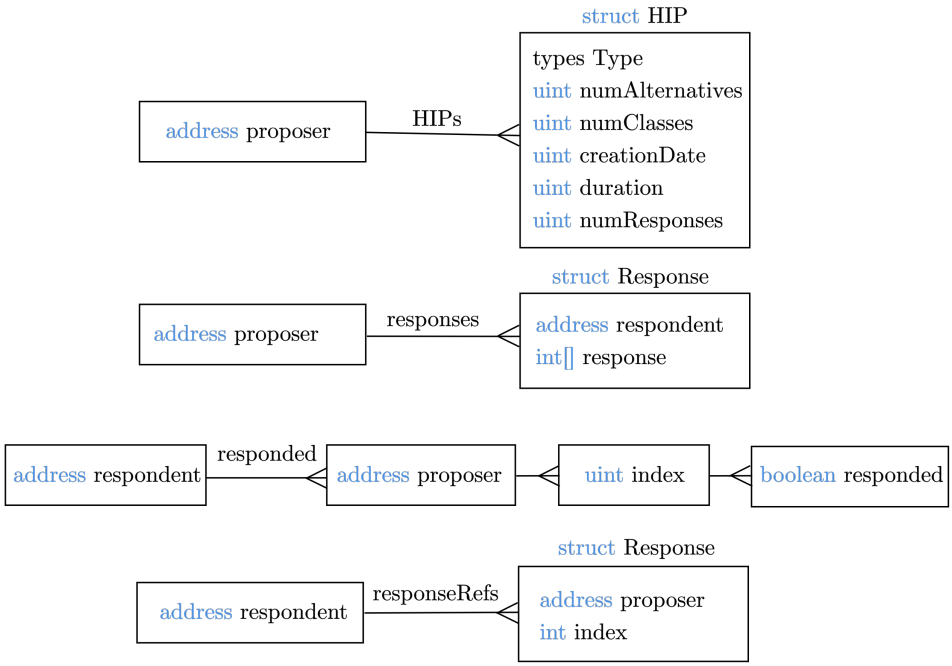

Figure 2: Main data structures

4.1 Response Verification

Before a response, submitted in the form of an array R of unsigned integers, is recorded in the contract, we must verify that it is valid for a given HIP, defined by a triplet (t, n, k).

- If t indicates a choice primitive, the validity requirements are that $length(R) == 1$ (i.e. we only allow singleton choices[2]) and $R[0] < n$ (i.e. the submitted choice corresponds to the index of a possible alternative).
- If t indicates a ranking primitive, the validity requirements are that $length(R) == n$ and R contains unique digits between 0 and $n - 1$. This latter requirement is verified by a function $uniqueDigits()$ in $O(n)$, which uses a local boolean array variable of size n. Depending on the preferred storage-computation trade-offs, an alternative would be to verify the uniqueness of the digits of R in $O(n^2)$, without the use of a local array. This is the most computationally-intensive potential operation in the proposed protocol.
- If t indicates a sorting or classification primitive, the validity requirements are that $length(R) == n$ and R only contains digits between 0 and $k - 1$.

4.2 Payment Processing

Compensating respondents to a HIP proportionally to its total number of respondents poses challenges for payment processing. It notably not allow for a real-time incrementation of a respondent's balance.

[2]This requirement could be changed to an inequality to allow for larger choice subsets.

This is due to the fact that any computation on the EVM must be initiated by an EOA, and it does not allow for automated code execution. The solution we propose is to compute this balance once a respondent requests a payment, using the `requestPayment()` function, so that they can bear the gas cost of this computation.

5 Example Applications

A wide range of tasks requiring human intelligence can be expressed as HIPs, for example within the training and operation of AI systems and the governance of blockchain systems and DAOs. Surveys can be modeled as an instance of P_1 (14), the collection of training data for machine-learned ranking (MLR) as P_2 (15), independent AI safety audits as P_3 (6), or data annotation as P_4 (16). In these examples, HIPs are used with a descriptive intent and a collection of individual preferences is their intended output. Moreover, when combined with a systematic aggregation procedure for individual preferences, HIPs can serve as primitives in processes such as plurality voting or approval voting as instances of P_1 (17), preferential voting as P_2 (18), score voting as P_3 (19), or rule-based classification as P_4 (20).

Following is an example, in pseudo-JavaScript, for a data annotation use case. The prefix `"contract."` indicates a call to a function of the contract by an EOA or a web client, e.g. using the *web3.js* library (21).

- Proposer calls a classification HIP with `await contract.methods.submitHIP(CLASSIFICATION, n, 2, duration).send({from:accounts[0], value:fee});`
- After a delay corresponding to the value of the argument `duration`, proposer collects responses with `response=await contract.methods.getResponse(proposer,index, i).call();`
- Proposer aggregates responses off-chain using e.g. the majority rule.

6 Conclusion and Perspectives

This paper described the design and implementation of an Ethereum protocol to to incentivize and coordinate collective intelligence by humans on-chain. Experiments using the proposed protocol will be conducted in the Algovera community, a DAO for data scientists, in order to identify new use cases and optimize gas usage for typical real-world applications.

The detailed source code of the proposed implementation can be found in the Appendix of this paper.

References

[1] Siddarth, D. *et al.* (2021) How AI Fails Us. Technology & Democracy Discussion Paper, Harvard Kennedy School, Carr Center for Human Rights Policy, Cambridge, Massachusetts.

[2] Ross, A. W. (1956) An Introduction to Cybernetics. London: Chapman & Hall Ltd.

[3] Licklider, J. C. R. (1960) Man-Computer Symbiosis, IRE Transactions on Human Factors in Electronics, 1, pp. 4–11.

[4] Xin, D. *et al.* (2018) Accelerating Human-in-the-loop Machine Learning: Challenges and Opportunities, DEEM'18: Proceedings of the Second Workshop on Data Management for End-To-End Machine Learning, June 2018, pp. 1-4.

[5] Wu, X. *et al.* (2022) A survey of human-in-the-loop for machine learning, Future Generation Computer Systems, 135, pp. 364-381.

[6] Falco, G. *et al.* (2021) Governing AI safety through independent audits, Nature Machine Intelligence, 3, pp. 566–571.

[7] Hagendorff, T. (2021) Blind spots in AI ethics, AI and Ethics, *Commentary*, pp 1-17.

[8] Hartvigsen, D. (2008) The Manipulation of Voting Systems, Journal of Business Ethics, 80 (1), pp. 13–21.

[9] Ebel, C. *et al.* (2021) Towards intellectual freedom in an AI Ethics Global Community. AI and Ethics, 1, pp 131-138.

[10] Halderman, J. A., Teague, V. (2015) The New South Wales iVote System: Security Failures and Verification Flaws in a Live Online Election, Proceedings of the International Conference on E-Voting and Identity, Lecture Notes in Computer Science, 9269, pp. 35-53.

[11] Little, G., Chilton, L. B., Goldman, M., Miller, R. C. (2009) TurKit: tools for iterative tasks on mechanical Turk. In Proceedings of the ACM SIGKDD Workshop on Human Computation (HCOMP '09), Paul Bennett, Raman Chandrasekar, Max Chickering, Panos Ipeirotis, Edith Law, Anton Mityagin, Foster Provost, and Luis von Ahn (Eds.). ACM, New York, NY, USA, pp. 29-30

[12] Nabben, K (2021) Imagining Human-Machine Futures: Blockchain-based 'Decentralized Autonomous Organizations', working paper, SSRN: https://ssrn.com/abstract=3953623.

[13] Roy, B. (1996) Multicriteria Methodology for Decision Aiding, Kluwer Academic Publishers, Dordrecht (1996).

[14] Rubenfeld, G. D. (2004) Surveys: An Introduction, Respiratory Care October, 49 (10), pp. 1181-1185.

[15] Rahangdale, A., Raut, S. (2019) Machine Learning Methods for Ranking, International Journal of Software Engineering and Knowledge Engineering, 29 (06), pp. 729-761.

[16] Paullada, A. *et al.* (2021) Data and its (dis)contents: A survey of dataset development and use in machine learning research, Patterns, 2 (11), 100336.

[17] Laslier, JF. (2012). And the Loser Is... Plurality Voting. In: Felsenthal, D., Machover, M. (eds) Electoral Systems. Studies in Choice and Welfare. Springer, Berlin, Heidelberg.

[18] Arrow, K. J. (1951) Alternative Approaches to the Theory of Choice in Risk-Taking Situations, Econometrica, 19 (4), pp. 404-437.

[19] Dery, L., Tassa, T., Yanai, A. (2021) Fear not, vote truthfully: Secure Multiparty Computation of score based rules. Expert Systems with Applications, 168, 114434.

[20] Li, X. L., Liu, B. (2014) Rule-based classification. In: Aggarwal CC (ed.) Data classification: algorithms and applications. CRC Press, Boca Raton, pp. 121–156.

[21] Lee, W.-M. (2019) Using the web3.js APIs, In: Beginning Ethereum Smart Contracts Programming, pp. 169–198, Apress, Berkeley, CA.

A Appendix: Source Code of the Proposed Protocol

```
// SPDX-License-Identifier: CC BY 4.0
pragma solidity ^0.8.12;
/**
* @title Human-augemented Intelligence contract
* @author Nassim Dehouche
*/
import "@openzeppelin/contracts/interfaces/IERC721.sol";
contract HaAI {
address owner;
address tokenContract ;
// HIP types
enum types{ CHOICE, RANKING, SORTING, CLASSIFICATION}
uint numProposers;
address[] proposers;
uint[] fees;

  constructor(){
    owner = msg.sender;
```

```
  }

  /**
  @param _tokenContract is the address of the ERC-721 contract to vet
  voters. We assume one address, one NFT, one vote.
  Use 0xF5b2B5b042B253323cB96121ABad487C95d287ea on Kovan
  */
function initialize (address _tokenContract, uint[] calldata _fees )
public{
    require(msg.sender == owner);
    tokenContract = _tokenContract;
    fees=_fees;
}

// The HIP structure
struct HIP{
  types HIPType;
  uint numAlternatives;
  uint numClasses;
  uint creationDate;
  uint duration;
  uint numResponses;
  }

// Mapping proposers with an array of their proposed HIPs
mapping(address => HIP[]) public HIPs;

// The Response struct for the content of the response.
struct Response{
  address respondent;
  uint[] response;
  }


// The Response reference struct for payment.
struct ResponseRef{
  address proposer;
  uint index;
  }

// Responses. The first key is the proposer address
mapping(address => mapping(uint => Response[])) internal responses;

// The Response boolean. The first key is the respondent address
mapping(address => mapping(address => mapping (uint =>bool))) public
    responded;

// The Response reference for payment. Mapping respondent with the
    HIPs they responded to.
mapping(address => ResponseRef[]) public responseRefs;

modifier onlyIfPaidEnough(types _HIPType) {
    require(msg.value==fees[uint(_HIPType)], "User did not pay the
        right fee for this HIP type.");
    _;
}

modifier onlyIfHoldsNFT(address _voter) {
    require(IERC721(tokenContract).balanceOf(_voter) > 0, "User does
        not hold the right NFT.");
    _;
}

modifier onlyIfHasNotResponded(address _proposer, uint _id) {
```

```
    require(responded[msg.sender][_proposer][_id]==false, "User has
        already responded.");
    _;
}

modifier onlyIfStillOpen(address _proposer, uint _id) {
    require(block.timestamp<=HIPs[_proposer][_id].creationDate+HIPs[
        _proposer][_id].duration, "This HIP is no longer open for
        responses.");
    _;
}

function submitHIP
(
  types _HIPType,
  uint _numAlternatives,
  uint _numClasses,
  uint _duration)
public
payable
onlyIfPaidEnough(_HIPType)

returns(uint _id)
{
bool condition;
if (_numAlternatives>=2){
   condition=true;
   if (_HIPType==types.SORTING || _HIPType==types.CLASSIFICATION){
    condition=_numClasses>=2;

    }
}

if(!condition) { revert('Trivial or invalid HIP'); }

_id= HIPs[msg.sender].length;
if (_id==0){
numProposers++;
proposers.push(msg.sender);
}
HIPs[msg.sender].push();
HIPs[msg.sender][_id].HIPType = _HIPType;
HIPs[msg.sender][_id].numAlternatives = _numAlternatives;
HIPs[msg.sender][_id].numClasses = _numClasses;
HIPs[msg.sender][_id].creationDate = block.timestamp;
HIPs[msg.sender][_id].duration = _duration;
return _id;
}

function rightDigits (uint[] calldata _response, uint _number)
internal
pure
returns(bool _right)
{
uint i;
_right=true;
while (i<_response.length){
 if (_response[i]>=_number){
   return false;
 }
 unchecked{i++;}
}
return _right;
}
```

```
function uniqueDigits (uint[] calldata _response, uint _number)
internal
pure

returns(bool _unique)
{
bool[] memory visited;
uint i;
_unique=true;
while (i<_response.length){
 if (_response[i]>=_number || visited[_response[i]]==true){
   return false;
 }
 else{
   visited[_response[i]]=true;
 }
 unchecked{i++;}
}
return _unique;
}



function submitResponse
(
  address _proposer,
  uint _id,
  uint[] calldata _response)
public
onlyIfHoldsNFT(msg.sender)
onlyIfHasNotResponded(_proposer, _id)
onlyIfStillOpen(_proposer, _id)
returns(uint _number)
{
bool condition;

    if (HIPs[_proposer][_id].HIPType==types.CHOICE){
   condition=_response.length==1 && _response[0]<HIPs[_proposer][_id].
      numAlternatives;
   }
    else if (HIPs[_proposer][_id].HIPType==types.RANKING){
    condition=_response.length==HIPs[_proposer][_id].numAlternatives
        && uniqueDigits(_response, _response.length);

    }
    else if (HIPs[_proposer][_id].HIPType==types.SORTING || HIPs[
        _proposer][_id].HIPType==types.CLASSIFICATION){
    condition=_response.length==HIPs[_proposer][_id].numAlternatives
        && rightDigits(_response, HIPs[_proposer][_id].numClasses);

    }


if(!condition) { revert('Invalid response'); }

_number=responses[_proposer][_id].length+1;
HIPs[_proposer][_id].numResponses=_number;
responses[_proposer][_id].push();
responses[_proposer][_id][_number-1].respondent=msg.sender;
 for(uint i = 0; i < _response.length; ) {
responses[_proposer][_id][_number-1].response.push(_response[i]);
unchecked{i++;}
}
ResponseRef memory r;
        r.proposer = _proposer;
```

```
        r.index = _id;

responseRefs[msg.sender].push(r);
responded[msg.sender][_proposer][_id]=true;
return _number;
}

// Respondents payment function
    function requestPayment() public
    {
    uint _balance;
    uint _id;
    address _proposer;
    for (uint i=0;i<responseRefs[msg.sender].length;){
    _proposer=responseRefs[msg.sender][i].proposer;
    _id=responseRefs[msg.sender][i].index;
    if (_proposer!=address(0) && block.timestamp>HIPs[_proposer][_id].
        creationDate+HIPs[_proposer][_id].duration)
    {
    responseRefs[msg.sender][i].proposer=address(0);
    _balance+=fees[uint8(HIPs[_proposer][_id].HIPType)]/HIPs[_proposer
        ][_id].numResponses;
    unchecked{i++;}
    }
    }
      (bool sent, ) = msg.sender.call{value: _balance}("");
        require(sent, "Failed to send Ether");

   }

 function getNumProposers() public view returns(uint _numProposers){
     return numProposers;
   }

 function getFee(uint i) public view returns(uint _fee){
     return fees[i];
   }
function getProposer(uint i) public view returns(address _proposer){
     return proposers[i];
   }

function getHIPCount(address _proposer) public view returns(uint
    _count){
     return HIPs[_proposer].length;
   }

function getResponse(address _proposer, uint _indexHIP, uint
    _indexResponse) public view returns(uint[] memory _response){
     return responses[_proposer][_indexHIP][_indexResponse].response;
   }

function getBalance() public view returns(uint _balance){

    uint _id;
    address _proposer;
    for (uint i=0;i<responseRefs[msg.sender].length;){
    _proposer=responseRefs[msg.sender][i].proposer;
    _id=responseRefs[msg.sender][i].index;
    if (_proposer!=address(0) && block.timestamp>HIPs[_proposer][_id].
        creationDate+HIPs[_proposer][_id].duration)
    {
    _balance+=fees[uint8(HIPs[_proposer][_id].HIPType)]/HIPs[_proposer
        ][_id].numResponses;
    unchecked{i++;}
    }
```

```
    }
    return _balance;
   }

}
```